REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
May 1990	Conference paper	

4. TITLE AND SUBTITLE

See report.

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

See report.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

See report.

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

See report.

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution Statement A - Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Presented at the IEEE 1990 National Aerospace and Electronics Conference (NAECON 1990) held in Dayton, Ohio, on 21-25 May 1990.

14. ABSTRACT

See report.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	UU		**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified			

Forward Chaining Parallel Inference[1]

Jay Labhart, Michael C. Rowe,
Merit Technology Incorporated
5068 W. Plano Parkway
Plano, Texas 75093-5009, (214) 248-2502

Steve Matney, and Steve Carrow
Naval Research Laboratory
4555 Overlook Ave. S.W.
Washington D.C. 20375-5000

Abstract

Rule based inference has demonstrated its applicability for a wide variety of domains. As users have grown more comfortable with this technology, the scope of attempted projects has grown from small laboratory demonstrations into massive real world time-critical systems. However, as the scope of systems has increased, execution speed has become unacceptable. One method of improving inference performance is parallelization. The parallelization of inference is not as straightforward as the parallelization of traditional numeric algorithms. Difficulties stem from the unpredictability of execution paths, small absolute task sizes, wide relative task size variances, and high proportion of shared volatile data.

This paper describes the completed and ongoing efforts of the Parallel Inferencing Performance Evaluation and Refinement project[1] (PIPER). PIPER Phase I produced an initial parallel inference engine (expert system tool kit) for the BBN Butterfly(R) Plus. Currently, PIPER Phase II is investigating parallel inference techniques on Thinking Machines' Connection Machine (CM) parallel computer.

The Phase I inference engine is based on the Merit Enhanced Traversal Engine (METE) algorithm which is an extension of Forgy's (1979) RETE algorithm. To evaluate the efficacy of this design and implementation, an iterating 108 rule knowledge base was composed. This rule set was designed to roughly simulate the information rich nature of its target application domain, Strategic Defense Initiative contact discrimination, and was processed on from 7 to 85 Butterfly(R) Plus processor nodes. Three uniprocessor control groups were also employed to gauge speed-up. Using the control group which produced the most conservative speed-up factors, the Phase I inference engine achieved a maximum true speed-up in excess of 29 utilizing. This speed-up was attributed to:

(1) parallelism in constant tests and two input node tests, as well as

(2) pipelining between
(a) two input node tests and conflict resolution, and
(b) test processing and overhead corresponding to the parallel implementation.

INTRODUCTION

Rule based inference has demonstrated its applicability for a wide variety of domains. As users have grown more comfortable with this technology, the scope of attempted projects has grown from small laboratory demonstrations into massive real world time-critical systems. However, as the scope of systems has increased, execution speed has become unacceptable. One method of improving inference performance is parallelization. The parallelization of inference is not as straightforward as the parallelization of traditional numeric algorithms. Difficulties stem from the unpredictability of execution paths, small absolute task sizes, wide relative task size variances, and high proportion of shared volatile data.

1. This work has been performed by Merit Technology, Inc. under contract N00014-88-C-2163 for the Naval Research Laboratory.

Inference Introduction

Expert systems are specialized problem solving computer software systems. A knowledge base contains (among other items) *facts* and *rules*. Facts and rules are processed by an inference engine to derive additional or modified facts. Rules, also called *productions*, generally take the form of *premise* and *action*. Production premises, also called *Left Hand Sides* (*LHS*), are composed of conjunctive and disjunctive *conditions*. When all of a rule's conditions are satisfied, the rule is eligible to execute or *fire* its *action clauses*. Action clauses, also called *Right Hand Sides* (*RHS*), of productions allow new facts to be asserted, existing facts to be modified and deleted, messages to be printed, and so on.

There are two major inference strategies, (1) *Forward Chaining* (data directed); and (2) *Backward Chaining* (goal directed). Forward chaining evaluates facts against rule premises. If a premise is true, then the production becomes eligible to fire. Rule firings produce additional facts or modify existing facts which may satisfy the LHS's of other rules. With additional facts, further production firings may occur. Forward Chaining continues until no additional productions can be fired (or in some systems, until a *goal* fact has been asserted). This paper discusses only the forward chaining inference aspects of PIPER.

Parallel Inference

The majority of inference tasks are small grained, in that they require as few as one to a couple of assembly instructions.[2] A few inference tasks are large grained, consisting of cross product operations requiring hundreds to thousands of assembly level instructions. The variance among task execution times is a limiting factor to total speed-up because execution time can be no less than the time necessary to run the longest task. Gupta proposed that speed-up can be estimated as a ratio of average to maximum task times.[2 3]

The success of parallel computation also relies on the even and efficient distribution of work among available processors (load balancing). *Compile Time* or *Static Load Balancing* can efficiently occur only when subtask use can be accurately

anticipated. This method introduces very little run-time overhead. Since inference paths cannot be predicted at compile time, this method of load balancing is not useful in this domain. *Dynamic load balancing* allocates work as it becomes available. This method is better suited to the inference problem in that it makes no assumptions about execution order. Dynamic load balancing can introduce significant run-time overhead. This overhead is a function of the size and number of subtasks that must be managed. As task size (grain size) decreases, the number of tasks that must be managed increases.

Inference overhead derives from four major sources:

- **Scheduling**: Resources needed by a parent task to notify the scheduler(s) that a new task has become available and the resources needed by the scheduler(s) to insert this new task into the task queue.
- **Dispatching**: Resources needed by the scheduler(s) and the available processor to transfer a task from the queue to the destination processor.
- **Results Management**: Resources needed to fan-out inference results to other processors. Typically, inference requires a large volume of global memory to maintain the constantly changing state of known facts.
- **Task Conclusion**: Resources needed to inform the scheduler(s) that a processor has become idle, and thus, can accept another task.

Many have investigated a wide range of parallel inference strategies. Forgy,[1] Gupta and Forgy,[3] Gupta,[2] Miranker,[4 5] Oflaser,[6] Kelly and Seviora[7] and others have investigated the parallelization of RETE derived algorithms. Cheng and Juang,[8] Conrey,[9] Conrey and Kibler,[10] Lin, Kumar and Leung,[11] Kumar and Lin[12] and others have studied methods of compile time partitioning of backward chaining and logic programming knowledge bases to exploit parallelism. Biswas, *et al.*[13], de Kergommeaux and Robert,[14] Kale'[15] and others have published detailed technical reviews and performed simulations of AND/OR parallelism in logic programming.

Butterfly(R) Parallel Computer

The BBN Butterfly(R) Plus computer consists of up to 256 processor nodes that are interconnected via a Butterfly(R) Switch. This switch is implemented by connecting 4-by-4 crossbar switches in the pattern similar to the butterfly transforms of FFTs. Each processor node contains a 16-MHz Motorola 68020 mpu, 68881 floating point processor, 68851 memory management unit, 4-Megabytes of local RAM, and microcoded processor node controller (PNC) for accessing the switch.

All memory in the Butterfly(R) Plus is shared, giving each processor access to a maximum, 1-Gigabyte of memory. Shared memory is accomplished by using the upper eight bits of the memory address to identify the memory's hosting processor node. When the memory management unit receives a request for a non-local memory access, it forwards this request to the PNC. The PNC generates a packet containing the specific request and sends it through the switch. If the remote memory request involves a read, then all processing is suspended on the local node until the memory request has been satisfied. Other operations, like writes, can be sent and monitored by the PNC without suspending other local processor node functions. The switch includes redundant paths between processor nodes, which allows the PNC to reroute packets from busy or disabled paths.

The major advantage of the switch interconnection strategy is that bandwidth can grow freely as additional processor are added. Thus, the switch does not have the saturation problem that is common to bus based architectures. The switch path length between any two processor nodes grows as a function of $Log_4(N)$, where N is the number of processor nodes. The major disadvantage of the switch is its speed — remote memory accesses can take up to 7000 nanoseconds vs 530 nanoseconds for local accesses.[16] Because of this local to remote memory access bias, experienced Butterfly(R) users pay special attention to minimizing remote memory reads.

The Butterfly(R) Plus computer runs the Chrysalis operating system, as well as the recently released Butterfly(R) MACH operating system. User application programs can be written in Butterfly(R) Fortran, Lisp, C and other programming languages. Shared memory access and management can be performed using programming languages (using 32 bit pointers) or via calls to the BBN Chrysalis operating system. Our experience found programming language shared memory management to be much faster than Chrysalis system calls. Unfortunately, programming language shared memory access proved extremely unreliable when used to communicate among more than two nodes of a heavily loaded system. The Chrysalis system calls proved absolutely reliable. We believe that this reliability difference relates to Chrysalis' integration of PNC microcode that handles local bus timings, switch packet collisions and packet retries.

PIPER is written in C running under the Chrysalis operating system and all shared memory access is managed with Chrysalis system calls.

THE METE ALGORITHM

The Knowledge Base Compiler

With the METE and PIPER inference technology, knowledge bases are compiled into a discrimination network called the METE net prior to inference. This compiler generates a processor-independent file containing test, memory, terminal, right hand side and back chaining nodes, in addition to data structures that are necessary to support the execution of forward and backward chaining inference. Nodes are linked to other nodes which serve as data token sources and sinks. During inference the METE net processes tokenized facts to recognize the satisfaction of rule LHSs.

METE Net Node Description

The METE net's test nodes correspond to rule LHS conditions. There are two types of test nodes, *Constant Test* (*CT*) and *TAND Test* nodes. CT test nodes contain a relational operator (*CT Operator*) and a *CT Value*. CT values can be a constant, an expression, or a pointer to a user defined function (such as a database , database query, or an expression. During inference, the CT value is compared (with respect to the CT operator) to the value of fact tokens. Figures 1 and 2 below contain a production and the corresponding compiled network, respectively. These figures will help illustrate the compilation process and relationship among the various node types.

The compilation of the production in Figure 1 will produce two CT paths. The first path (see Figure 2) will recognize instances of this specific *Target* and the second path will recognize instances of this specific *Aircraft*. The heads of the object paths are hashed memory addresses. Thus, if a fact token type was for *Target* objects, the token would be forwarded to the *Target* path. In sequence, this *Target* head node is followed by CT nodes that recognize attributes of *Type* equal to "*YAK-38*", *Range* less than "*1000*" nautical miles and *Bearing* equal to

"*Closing*" If a token fails any CT test, then it is absorbed and will not reach the next test.

```
RULE-1:
IF Target  WITH          /* IF there is an instance of object-type TARGET  */
     Type = YAK-38  AND  /* WITH attribute TYPE = YAK-38                   */
     Range < 1000  AND   /* AND attribute RANGE less than 1000
nm                                                                         */
     Bearing = Closing AND  /* AND attribute BEARING = Closing
AND                                                                        */

   Aircraft  WITH        /* an instance of object-type AIRCRAFT            */
     Type = F-14  AND    /* WITH attribute TYPE = F-14                     */
     Weapon = Guns       /* AND attribute WEAPON = Guns                    */

THEN            /* THEN CREATE an instance of Object STATUS              */
  ASSERT Status WITH    /* WITH attribute CONDITION = Yellow              */
              Condition = Yellow.
```

Figure 1: Sample Rule to Compile.

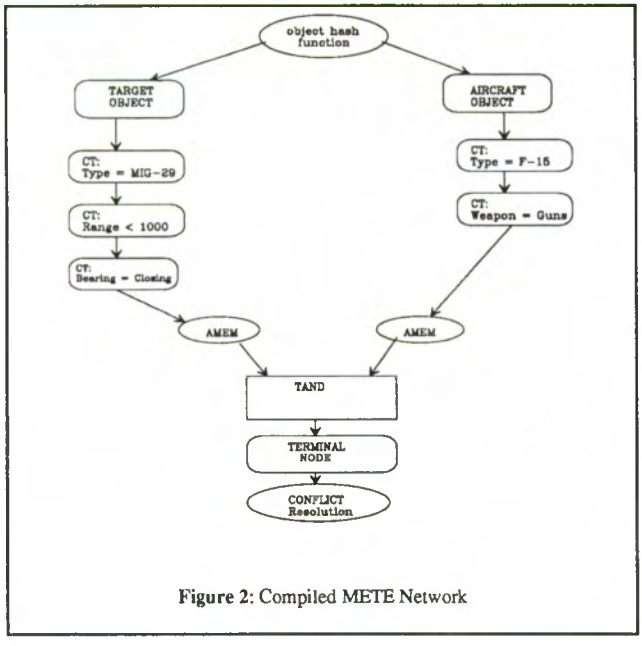

Figure 2: Compiled METE Network

Fact tokens that pass all tests in a CT path are stored in a memory node called an **AMEM**[2]. Each CT path has a distinct AMEM structure to maintain the set of successful tokens. In this example, there may be multiple *Target* objects (*T1* and *T3*) and *Aircraft* (*A1* and *A6*) that satisfy the attribute conditions of the CT test paths for *Target* and *Aircraft*, respectively. The *Target* AMEM will contain references to the *Target* object instances *T1* and *T3*. Tokens and AMEM objects reference a global structure called the *Working Memory*. The Working Memory maintains all facts that are currently known to the system.

The other kind of test nodes, TAND[3]s, perform *Unification* between two memory nodes to determine if they contain tokens that satisfy and are consistent with dependent premise conditions. In the case of the current example, the TAND node would check the Target and Aircraft AMEMs. If both of these AMEMs contain tokens, then the TAND node can perform a join of the two sets and pass the resulting joined tokens to a *Terminal Node*. With the tokens that have already been mentioned, the Terminal node would receive the join of the successful *Aircraft* and *Target* objects ({*A1 T1*, *A1 T3*, *A6 T1*, and *A6 T3*}). These four tuples represent four satisfactions of example rule 1. Tokens that do not meet the constraints of a TAND node are absorbed and do not progress to subsequent nodes.

TAND nodes contain a binary Boolean operator, the *TAND Operator*. A diversity of TAND operators allows for an extensive variety of possible object relations. The current example uses the *AND* operator. To handle increased unification complexity for productions that contain more than two dependent conditions, a tree is constructed of multiple TAND nodes. Successful tokens cascade through this tree. Increasingly complex relations can be achieved by increasingly deeper TAND trees. Between successive TAND nodes are *SMEMs* (Smart MEMorys). A SMEM contains the reference sets of tokens that passed the preceding TAND nodes' tests and serves as the input buffer for the subsequent TAND node.[4]

Terminal nodes are the culmination of production premises. Terminal nodes reference the immediately preceding SMEM which contains references to the set of joined tokens, each set satisfies all conditions of the production. A Terminal node also contains references to RHS Nodes and to *Backward Chaining Nodes* (*BCON Nodes*). When executed, the RHS Nodes modify Working Memory by *asserting* (creating new objects), *deleting* existing objects, or *modifying* attributes of existing objects. BCON Nodes reference AMEM structures that contain object references that are necessary to prove sub-goals needed to support backward chaining inference.

The Inference Cycle

The METE algorithm employs a three stage inference cycle similar to the RETE algorithm. The first stage, *Match*, reviews existing facts to determine which productions are eligible to fire. The second stage, *Select*, picks the highest priority, eligible production(s) (maintained in a structure called the *Activation List*) to fire based on a selection strategy (See Brownston, et al.[17] for a discussion of conflict resolution methods). If no productions are eligible to fire, then inference is complete. The final stage, *Act*, fires the selected production(s) and updates the working memory structure. Following the Act stage the next inference cycle begins with the start of a Match stage.

The PIPER parallel-METE algorithm exploits multiple forms of parallelism. This parallelism is made possible by partitioning processors into three functional sets. The first set consists of a single processor called the *Inference Manager* (*IM*). The other two sets perform functions related to the node types that they contain, Constant test nodes (*CT processors*) and TAND test nodes (*TAND processors*). The IM maintains a Chrysalis operating system *dual queue* between itself and each of the CT and TAND processors. The Chrysalis dual queue use a PNC microprogrammed function to maintain a message queue between two Butterfly(R) processors. See Figure 3.

Each CT processor contains a copy of all CT test nodes, and thus, any CT processor can perform the constant tests on any token that it receives. The IM begins each inference cycle by distributing new fact tokens (using a round-robin scheme) to CT processors. CT processors acknowledge (to the IM) the completion of each token's processing. Tokens that meet the conditions of a CT path are forwarded to a distinct TAND

2. The name AMEM derives from the first (or alpha) memories that were use in the original RETE net algorithm developed by Forgy[1]

3. For simplicity, we call all two input unification nodes, regardless of node operator type, TAND nodes. Typical knowledge bases use far more *AND* node operators than all other operators combined.

4. It is relatively rare for a production to consist of only a single object. In this case, there is no need for a TAND node and therefore a path can be constructed directly from a CT to AMEM to Terminal node.

cycle_counter value, two different *tester* objects' *type* attribute (namely the *I* and *J* of the rule label), and matched 2 *tester* objects' *slot* values. These rules required 3 CT tests and 2 TAND tests to implement. Each rule satisfaction created two *tester* facts of identical *types* (say *I* and *J*) to the *tester* facts that satisfied their LHSs. These new *tester* facts were used in the next iteration to completely or partially satisfy the rules containing *I*'s or *J*'s in their rule labels. The interaction of these rules and *tester* facts created a rapid explosion of rule satisfactions and new fact assertions. Each of these rules fired 1 time the first iteration and 10 times the second iteration. The number of rule firings would have continued to grow at a rate of one magnitude per iteration. Fact growth was two times that of rule firings since each rule produced two new facts.

- 1- typo_body rule. This rule was labeled *rule_I_2* and was intended to be the sixth body rule, but because of a typo this rule contained one additional condition. This rule executed once in each of the first and second iterations and produced two facts. This rule required the satisfaction of 4 CT tests and 2 TAND tests to fire.

- 100- body_filler rules. These rules were labeled *rule_X_J* (where *X* is and alpha character from '*a*' to '*x*' and *J* varied from 0 to 4) and were identical to *body* rules except that they tested *cycle_counter value* and *tester slot* values that were never satisfied. Thus, facts failed at the top level CT tests. These rules added a great deal of additional processing load.

To summarize the benchmark processing scenario, six initial facts were asserted. These initial facts resulted in a total of 59 (7 first iteration and 52 second iteration) rule firings, which produced 114 (12 first iteration and 102 second iteration) new fact assertions, 1 fact modification (during the first iteration) and one fact deletion (during the second iteration). Of the 100 rules that were never satisfied, each failed objects at their top level CT nodes. This knowledge base produced a testing load similar to a much larger knowledge base.

The Experimental Design

This knowledge base was executed under several conditions on the NRL 128 node Butterfly(R) Plus computer and also on a SUN 3/60 UNIX(R) workstation. The experimental conditions were grouped along two dimensions,

(1) Total number of processors utilized, and

(2) Ratio of CT to TAND processors. The current design used only a single IM processor, regardless of the number of CT or TAND processors.

Since this design assigned distinct activities to three different processors classes, varying the ratio of processors assigned to these classes yielded an understanding of their interrelation. Levels of processor numbers included: 1 (PIPER uniprocessor and MeriTool[tm] serial versions), 7, 13, 19, 25, 31, 37, 43, 49, 55, 61, 67, 73, 79 and 85 processors. Number of processors 7 through 85 allowed the testing of even ratios of CT to TAND processors corresponding to the ratios of 1:5, 2:4, 3:3, 4:2, and 5:1. Total execution time was the dependent variable.

The assignment of functions to specific processors can affect the communication loading at individual switch elements. Switches that are are more heavily loaded have higher incidences of packet collisions and associated message delays. To balance the consequences of node allocations, a complete and randomly ordered presentation of all possible experimental conditions was attempted on three different days. Because of a condition (known as *node rot* among NRL Butterfly users) not all of the conditions could be completed. The 3:3 ratio for 85 processors was never completed, the other ratios of 85 processor conditions were only completed once, and the 4:2 ratio of 76 processors was completed twice.

The Benchmark Control Conditions

Three different control conditions were evaluated in this study. These control conditions included:

(1) The C language version of the commercial product form of MeriTool[tm], with as few modifications as possible. Changes were needed to conform with Chrysalis conventions of heap memory allocation. This version was run on a single node of the NRL Butterfly(R) Plus.

(2) The C language version of the commercial product form of MeriTool[tm], without modification, run on a SUN 3/60 workstation owned by Merit Technology.

(3) A PIPER uniprocessor version, containing only minor changes to the parallel PIPER version. These changes supported dual queues from and to the single processor.

THE BENCHMARK RESULTS

The Control Conditions' Results

MeriTool[tm] serial ran the BOGUS.KB in 6.7 seconds on the SUN 3/60 and 7.3354 seconds on a single node of the Butterfly(R) Plus. Both of these computers are based on MC68020 microprocessors and have other similar Motorola 68XXX chip set components. The two major differences are clock speed and operating systems; the SUN 3/60 CPU is clocked at 20 MHz while the Butterfly(R) Plus CPUs are clocked at 16 MHz. When the Butterfly(R) Plus node clock speed was normalized to that of the SUN's 20 MHz, its execution time would be 5.87 seconds[5]. From these results one can see that the Butterfly(R) version of MeriTool[tm]'s execution time on a single node was representatively similar to that of the commercial SUN version of MeriTool[tm]. See Table 1 below for a summary of these results.

The PIPER uniprocessor version ran considerably slower (13.0679 seconds, 9.80 seconds normalized to 20 MHz) than the Butterfly(R) MeriTool[tm] version. The authors attribute much of this performance lag to Chrysalis overhead related to dual queue message transmissions. We suspect that even though the single processor version of PIPER was sending messages to itself, the messages went through the PNC, through the Butterfly(R) Switch and back to the PNC. Thus, in the uniprocessor PIPER version, communications took the long way around and the PNC probably got in the way of itself, creating contention for the switch.

Gupta[2] reported results from simulation of uniprocessor and mulitporocessor environments for several historically significant rule sets. These simulations were based on timing Rete net code segments. Because these results do not include execution of fully functional code, absolute time comparisons between PIPER and Gupta's results can not be made. Gupta[2] reported both *true speed-up* and *nominal speed-up* times. *True speed-up* used the execution time of an efficient uniprocessor model (without any of the parallel synchronization or overhead) as the speed-up basis; thus, *true speed-up* represents the speed advantage between an efficient uniprocessor and an efficient

5. This normalization may not be entirely valid as it assumes all resources are scaled at the same rate as the microprocessor clock.

processor. This CT to TAND communication is implemented using Chrysalis system dual queues.

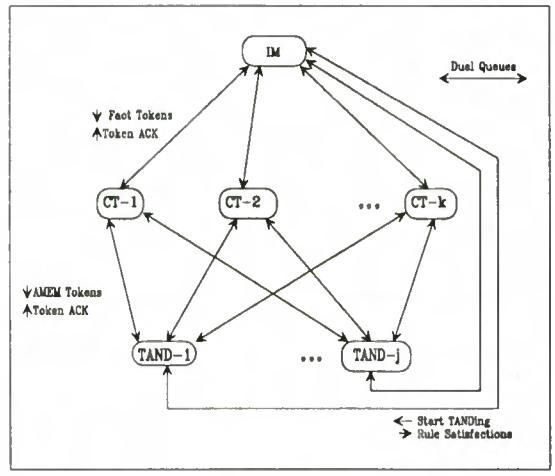

Figure 3: PIPER Parallel Inference Architecture

Processing multiple tokens through the CT portion of the METE net represents one form of parallelism. CT tests tend to be more homogeneous in processing time than does the pool of CT and TAND processing tasks. The more homogeneous the processing time for CT tasks, the better this simple but efficient load balancing strategy will be.

Tokens that are received by TAND processors from CT processors are immediately dequeued and placed in appropriate AMEM structures. This queuing and dequeuing is potentially a significant overhead source. But, since the queuing by a CT processor occurs in parallel with other CTs' token processing and the TAND token dequeuing is occurring simultaneously with CT processing, this potential for overhead is minimized. Each TAND processor is specialized, containing only specific TAND tests. The top level of TAND nodes are associated with a left and right hand AMEM structure. TAND nodes test a binary Boolean relationship between the "*m*" left hand and the "*n*" right hand AMEM entries. Thus, although this process is of *O(m*n)* complexity, each iteration is extremely simple computationally, being primarily limited by memory bandwidth.

Various investigators have studied the possibility of performing asynchronous TAND testing (testing as each AMEM entry arrives; Gupta[2] provides a fine analytical review of techniques). The major difficulty with completely asynchronous processing centers around the added overhead involved in performing cross product type operations on two simultaneously volatile sets. Both left and right hand TAND node input sets are subject to simultaneous additions and deletions. Considering the computational simplicity of the Boolean tests that are being performed in this cross product operation, it is not surprising that the overhead of completely asynchronous TAND token processing can easily overcome any benefit.

When the PIPER IM receives the acknowledgment that the last CT token has been processed, it transmits a TANDing start-up token to each of the TAND processors. Parallel TAND processing is a second form of parallelism.

Terminal nodes send Activation List updates to the IM via dual queues. The IM processes these updates as they are received using one of the user specified conflict resolution strategies. Conflict resolution involves sorting rules that are eligible to fire based on a priority scheme. Thus, the TAND processing of the Match phase is pipelined to the conflict resolution of the Select phase. This represents a third major source of parallelism. Once all TAND processing is completed, the IM can immediately start the Act phase.

With the selection of a production(s) to fire, the Act Stage begins. Each processor maintains a copy of working memory facts. In the ACT stage, the IM distributes working memory update information to all CT and TAND processors using the same dual queues as the Match phase. Working memory updates result from the execution of the RHS(s) of a rule(s). Additionally, a list of working memory changes, in the form of fact tokens, is produced. This list of fact tokens is processed by the CT processors in the next Match stage to determine which additional rules become eligible to fire. Since the dual queues maintain order of their entries, the IM does not have to wait for the ACT stage to complete before passing out the fact tokens for the Match stage. Thus, the Act and the Match stage are pipelined. This pipelining minimizes communications overhead of fact token fan out and working memory updating.

THE BENCHMARK METHOD

The Benchmark Knowledge Base

To estimate the efficacy of the PIPER parallel method, an iterating 108 rule knowledge base was composed (we call it BOGUS.KB). This knowledge base was designed to roughly simulate the fact-rich environment of PIPER's target domain, SDI contact discrimination. The knowledge base contained three object types:

- tester — which had two attributes, *type* and *slot*. There were four initial assertions of this fact that varied the value of the the *type* attribute from 0 to 3. The *slot* attribute of each fact was initially set to 0 and was tested and incremented by ten in each inference cycle.
- cycle_counter — had a single attribute, *value*, that was initialized to 0 and was incremented by 1 in each inference cycle. Each rule required the existence of this fact to fire.
- max_cycle — had a single attribute, *count*, that was initialized to 1 and was tested against the *value* attribute of the *cycle_counter* object. Thus, two complete inference cycles were performed (0 and 1). When the two attributes, *count* and *value*, were equal, the *cycle_counter* object was deleted, which blocked all future rule satisfactions.

The knowledge base contained the following numbers and types of rules:

- 1- **cycle_counter rule** incremented and tested the *cycle_counter value* attribute in all but the last iteration. If "*k*" inference cycles were run, this would fire "*k-1*" times.
- 1- **stopper rule** executed the last iteration in place of the *cycle_counter* rule and deleted the *cycle_counter* fact which made the truth value of the other rules false.
- 5- **body rules**. These rules labeled *rule_I_J* (where *I* and *J* varied from 0 to 3, except *rule_1_2* which is described immediately below). Each of these rules tested the

parallel execution of a rule set. *Nominal speed-up* used a uniprocessor execution time (including the unneeded synchronization and overhead added for parallelism) as the speed-up basis. Thus, *nominal speed-up* corresponds to the difference between a less than optimal uniprocessor and an efficient parallel execution.

Table 1: Benchmark Control Groups' Results

CONTROL GROUP	EXECUTION TIME
Sun 3/60 MeriTool[tm]	
20 MHz	6.7 seconds
Butterfly[(R)] MeriTool[tm]	
16 MHz	7.3354 seconds
20 MHz (normalized)	5.87 seconds
PIPER Butterfly[(R)] Uniprocessor	
16 MHz	13.0679 seconds
20 MHz (normalized)	9.80 seconds

The data from all three control groups yielded interesting results when they were used as the basis value of speed-up. The speed-up functions of the three are multiplicatively related and vary as a ratio of their basis groups' speeds. For instance, since the uniprocessor PIPER version had the longest execution time, it greatly inflated speed-up. This uniprocessor PIPER version corresponds to the basis value of Gupta's *nominal speed-up* reports. For the remainder of the results discussion, the MeriTool[tm] Butterfly[(R)] version was used as the basis. It is the authors' opinion that this MeriTool[tm] Butterfly[(R)] version's performance most fairly represents the speed-up a user could anticipate going from a 16 MHz MC68020 based uniprocessor computer to a 16 MHz MC68020 based parallel processor (like the Butterfly[(R)] Plus). This uniprocessor MeriTool[tm] Butterfly[(R)] version corresponds to the basis value of Gupta's *true speed-up* reports.

The Experimental Conditions' Results

Using the Butterfly[(R)] Plus MeriTool[tm] version (16 MHz) as a basis, a maximum single run speed-up of 31.59 was observed with 49 nodes (40 CTs and 8 TANDs). The maximum mean speed-up over three trails was 29.17 with 61 processors (50 CTs and 10 TANDs). The six fastest mean speed-ups all occured within the 5:1 (CT:TAND) ratio (see Figure 4).

The mean, standard deviation of the within experimental condition speed-ups (using only conditions with three completed observations) was 0.602. This indicates that an experimental condition's speed-ups could be expected to vary within a range of plus or minus 0.602 of its mean about 68 percent about of the time.

The ratios of 5 CTs to 1 TAND and then 4 CTs to 2 TANDs consistently out performed the other ratios. Over 75% of the maximum achieved speed-up was gained with the first 25 processors. After the employment of 25 processors the gain in speed gradually slowed. Beyond 55 processors all ratios demonstrated slowing and in some cases small negative speed-ups.

DISCUSSION OF THE RESULTS

This total speed-up was attributed to

(1) Parallelism in constant tests and two input node tests, as well as

(2) Pipelining between
(a) Two input node tests and conflict resolution, and

(b) Test processing and overhead corresponding to the parallel implementation.

To gain a detailed understanding of these results, one must evaluate the amount and type of work available, the number and type of processors available to perform the work, in addition to the overhead functions. Although time did not permit a detailed empirical study of this matter, below is one set of likely hypotheses.

In the second iteration, where the majority of the facts were processed, 102 facts went through CT testing. These facts only completely supplied tokens (to both left and right hand AMEMs of TANDs) to AMEMs of seven distinct TAND sub-trees. Of these TAND sub-trees, only the five *body* rules required significant processing time. Thus, at most, five TAND processors were heavily employed, while any number of TAND processors over five, added little useful processing (in the case of *increment* and *stopper* rules) or no work (in the case of the 100 *body_filler* rule). TAND processors that were not usefully employed still required resources of the IM processor during the act phase of the inference cycle and in synchronization between CT testing and TAND testing sub-phases of the match phase.

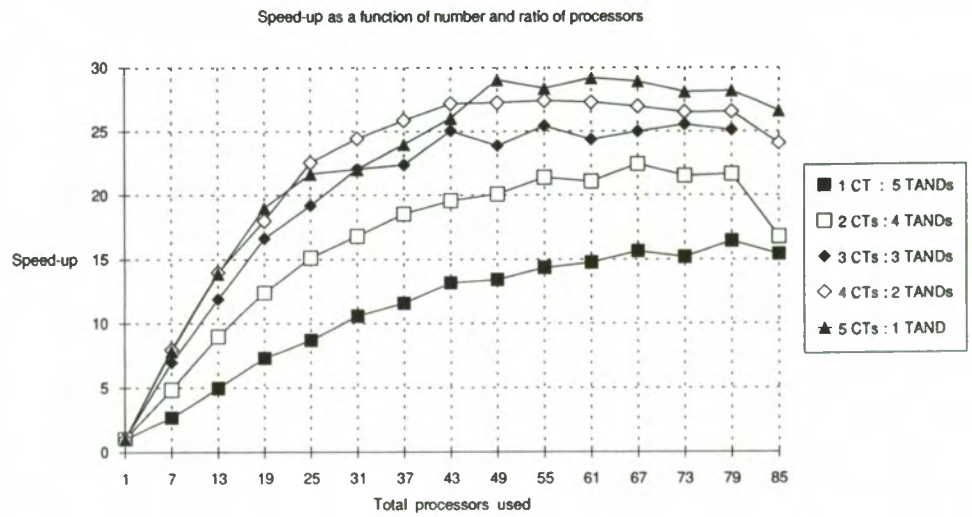

Figure 4

Thus, for this specific two iteration benchmark using the BOGUS.KB rule set, some number between 5 and 7 TAND processors would probably be optimal.

Similarly, the inference engine performed CT testing of at most 102 distinct facts in an inference cycle. As the reader will recall from a previous section on the design of the inference engine, CT testing is much less computationally demanding than TAND testing. The benchmark results are consistent with the hypothesis. It seems that by the time the IM had distributed several tokens for CT testing (say about 50) the first CT processors that received tokens had completed their testing and then sat idle. For this benchmark, any more than approximately 50 CT processors did not seem to benefit overall processing speed. If additional iterations of the knowledge base inference cycle could have been run[6], additional CT processors most likely could have been usefully engaged. Also, if the initial scheme of using lower level communications (rather that Chrysalis dual queues) would have been technically feasible, then the IM would have been able to disperse tokens faster (an estimated 4 to 10 times faster). Thus, additional CT processors could have probably been successfully employed. For this specific two iteration benchmark using the BOGUS.KB rule set, some number between 40 and 50 CT processors would probably be optimal.

Because the maximum speed-ups were recorded at the highest ratio of CTs to TANDs and also to determine if our hypotheses about CT to TAND ratios were correct, we executed several after the fact tests employing higher ratios. Basically, 43, 49 and 55 total processors with from one to seven TANDs (the remainder of the processors were assigned to an IM and the rest to CT processors). The maximum speed-up observed in these sessions was 26.19 at 49 total processors configured as 1 IM, 42 CTs, and 7 TANDs. These data do not completely endorse our conclusions as we would have expected the 42 CTs to 7 Tands or the 47 CTs to 7 Tands ratios to actually out perform all of the original experimental conditions.

CONCLUSIONS

Using a completely different mix of speed-up techniques, PIPER's actual results compare favorably to those of Gupta's simulations. When all expected overhead sources were added to Gupta's simulations (pages 140-144)[2] an average *true speed-up* across his six rule sets of 9.29 was achieved (with the single best rule set achieving a 13 fold speed up). To reach this performance, Gupta employed the following features:

* Hardware task schedulers that minimized the overhead needed to support optimized load balancing;

* Intra-node parallelism which reduced the impact of cross-product TAND-type testing; and

* Action parallelism which processed the working memory changes of the ACT phase in parallel.

Perhaps one reason for PIPER's successful speed-up is the nature of its benchmark rules set. Most knowledge bases are by design explicitly serial. The domain which the BOGUS.KB roughly simulates is inherently information rich, and thus, it more easily utilizes parallel processing techniques. The authors strongly believe that there are many other real world domains with similar fact-rich processing characteristics. A few examples of these domains include:

* Strategic Defense Initiative categorization and discrimination tasks.

* Rule based event simulations.

* Rule based sensor fusion and threat avoidance.

* Real time factory control and process monitoring.

* Intelligence analysis and data fusion.

Gauging from the high level task analysis, the performance functions seem to be moderately well behaved. Performance modeling and prediction is one of the topics that we are investigating. This topic can be subdivided in to two related areas:

* predict inference performance given a hardware configuration and rule set, or

* predict optimal configuration for a given rule set.

The Butterfly architecture is burdened with an interconnection topology that is more suited to large grained parallelism. We are currently investigating implementation techniques for a Connection Machine. At this point the Connection Machine seems better suited to the very fine grained parallelism of inference tasks.

BIBLIOGRAPHY

1. Forgy, C.L., "On The Efficient Implementation of Production Systems", *Ph.D. Thesis*, Camegi-Mellon University Department of Computer Science, 1979.

2. Gupta, A., *Parallelism in Production Systems*, Morgan Kaufmann, 1987.

3. Gupta, A. and C.L. Forgy, "Measurements on Production Systems", *CMU-CS-83-167*, 1983.

4. Miranker, D.P., "TREAT: A New and Efficient Match Algorithm for AI Production Systems", *UT Austin TR-87-03*, 1987.

5. Miranker, D.P., "TREAT: A Better Match Algorithm for AI Production Systems", *UT Austin TR-87-58*, 1987.

6. Oflazer, K., "Partitioning in Parallel Processing of Production Systems", *Carnegie Mellon University TR CMU-87-114*, 1987.

7. Kelly, M.A. and R.E. Seviora, "A Multiprocessor Architecture for Production System Matching", *AAAI-87*, 1987.

8. Cheng, P.D. and J.Y. Juang, "A Parallel Resolution Procedure Based on Connection Graph", *AAAI-87*, 1987.

9. Conery, J.S., "The AND/OR Process Model for Parallel Interpretation of Logic Programs", *University of California, Irvine TR 204*, 1983.

10. Conery, J. S. and Kibler, D. F., "AND Parallelism and Nondeterminism in Logic Programs", *New Generation Computing*, vol.3, 1985.

11. Lin,Y.J., Kumar,V. and Leuny,C.,"An Intelligent Backtracking Algorithm For Parallel Execution Of Logic Programs", *UT Austin TR 86-22*, 1986.

12. Kumar, V. and Lin, Y.J.,"A Framework for Intelligent Backtracking in Logic Programs", *UT Austin TR 86-36*, 1986.

13. Biswas, Prasenjit, Su, S.C., and Yun, D.Y.Y., "A Scalable Abstract Machine Model to Support Limited-OR (LOR) / Restricted-AND Parallelism (RAP)", *Proceedings of the Fifth*

6. The third iteration exhausted memory resources of the Butterfly(R) Plus. This iteration attempted to assert in excess of 1000 additional facts.

International Logic Programming Conference, The MIT Press, 1988.

14. de Kergommeaux, C. and Robert, P. "An Abstract Machine to Implement Efficiently OR-AND Parallel Prolog", *Proceedings of the Fifth International Logic Programming Conference*, The MIT Press, 1988.

15. Kale', L.V., Ramkumar, B., and Shu, W. "A Memory Organization Independent Binding Environment for AND and OR Parallel Execution of Logic Programs", *Proceedings of the Fifth International Logic Programming Conference*, The MIT Press, 1988.

16. BBN 1988, *The Butterfly(R) GP1000 Switch: Design and Function*, BBN Advanced Computers Inc., 1989.

17. Brownston, L., Farrell, R., Kant, E., and Martin, N., *Programming Expert Systems in OPS5*, Addison-Wesley, 1985.

TRADEMARKS

Butterfly(R) is a registered trademark of BBN Advanced Computers, Inc.

MeriTool(tm) is a trademark of Merit Technology Incorporated.

UNIX(R) is a registered trademark of AT&T.